Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Joni Tefke

Progressiivisten verkkosovellusten välimuistin käyttöstrategiat

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

11.3.2020

Tekijä Otsikko Sivumäärä Aika	Joni Tefke Progressiivisten verkkosovellusten välimuistin käyttöstrategiat 28 sivua 11.3.2020
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Mobile Solutions
Ohjaaja	Yliopettaja Hannu Markkanen

Insinöörityön tarkoituksena oli tutustua progressiivisiin verkkosovelluksiin ja niiden käyttämiin teknologioihin. Lisäksi työssä perehdyttiin erilaisiin välimuistin käyttöstrategioihin, joiden avulla sovellus saadaan toimimaan tehokkaasti huonon verkkoyhteyden alueella ja täysin verkottomassa tilassa. Välimuistin käyttöstrategioiden havainnollistamisen tueksi luotiin prototyyppisovellus, jonka avulla esiteltiin erilaisten strategioiden käyttöä.

Tietoa eri teknologioista ja välimuistin käyttöstrategioista haettiin suurimmaksi osaksi verkosta, koska ne ovat niin uusia, että niistä ei vielä löydy sopivaa kirjallisuutta. Prototyyppisovellus luotiin React-JavaScript-kirjastoa käyttämällä, ja käyttöliittymän luomiseen käytettiin Material-UI-viitekehystä. Progressiivisen verkkosovelluksen todentamiseen käytettiin Lighthouse-työkalua.

Lopputuloksena insinöörityössä saatiin luotua prototyyppisovellus, joka toimii tehokkaasti hitaan verkkoyhteyden alueella ja täysin verkottomassa tilassa sekä täyttää Lighthouse-työkalulla tehdyn testauksen perusteella kaikki progressiivisen verkkosovelluksen kriteerit. Sovellus yhdistelee erilaisia välimuistin käyttöstrategioita suorituskyvyn maksimoimiseksi. Strategiat valittiin jokaiseen tapaukseen erikseen siten, että käyttäjäkokemus ei kärsi ja sovelluksen suorituskyky pysyy ennallaan.

Insinöörityöhön kerättyjen tietojen avulla on mahdollista saavuttaa perustietämys progressiivisten verkkosovellusten teknologioista ja vaatimuksista. Lisäksi työn avulla voidaan tehostaa ja nopeuttaa progressiivisten verkkosovellusten välimuistin käytön suunnittelua ja toteuttamista.

Avainsanat	progressiivinen verkkosovellus, välimuisti, mobiilisovellus

Author Title Number of Pages Date	Joni Tefke Caching strategies of the progressive web applications 28 pages 11 March 2020
Degree	Bachelor of Engineering
Degree Programme	Information and Communication Technology
Professional Major	Mobile Solutions
Instructor	Hannu Markkanen, Principal lecturer

The purpose of this final year project was to familiarize oneself with the progressive web applications and technologies used by them. Also, the project aimed to dive more deeply into caching strategies and ways to make the application work with a poor network connection and offline mode. Finally, the goal was to create a prototype application that demonstrates the usage of the caching strategies.

Information on technologies and caching strategies was mostly retrieved from the internet, because they are so new that no suitable literature is yet available. The prototype application was created using the React JavaScript library and the Material-UI framework was used to create the user interface. The Lighthouse tool was used to verify the criteria of progressive web application.

The final result of this thesis was a fully functioning prototype application that works efficiently with a slow network and completely offline mode and fulfills all criteria of a progressive web application based on testing with Lighthouse. The application combines different caching strategies to maximize performance. Strategies are selected on a case-by-case basis so that the user experience is not compromised, but application performance remains unchanged.

With information collected in this thesis, it is possible to gain a basic understanding of the technologies and requirements of progressive web applications. In addition, this work can enhance and speed up the design and implementation of caching for progressive web applications.

Keywords	progressive web application, cache, mobile application

Sisällys

Lyhenteet

1 Johdanto 1

2 Progressiiviset verkkosovellukset 1

2.1 Määritelmä ja vaatimukset 1
2.2 Sovelluskuori 3
2.3 Service worker -teknologia 5
2.4 Web App Manifest -tiedosto 7
2.5 Taustasynkronointi 8
2.6 Push-ilmoitukset 9

3 Välimuistin käyttöstrategiat 11

3.1 Vain välimuisti -strategia 11
3.2 Välimuisti, varalla verkko -strategia 12
3.3 Välimuisti varmistaa datan käyttökelpoisuuden -strategia 12
3.4 Verkko, varalla välimuisti -strategia 13
3.5 Ensin välimuisti, sitten verkko -strategia 14
3.6 Käyttäjä kontrolloi välimuistia -strategia 15
3.7 Push-ilmoituksen yhteydessä -strategia 16
3.8 Taustasynkronoinnin yhteydessä -strategia 17
3.9 Muiden kuin GET-verkkopyyntöjen kontrollointi 18

4 Prototyyppisovellus 18

4.1 Käytetyt teknologiat 19
4.2 Välimuistin käyttö 20
4.3 Testaus 25

5 Yhteenveto 27

Lähteet 28

Lyhenteet

API	Application Programming Interface, ohjelmointirajapinta.
CSS	Cascading Style Sheet, verkkosivun tyyliohje.
HTTP	HyperText Transfer Protocol, protokolla, jota selaimet ja WWW-palvelimet käyttävät tiedonsiirtoon.
HTTPS	HyperText Transfer Protocol Secured, protokolla, jota käytetään tiedon suojattuun siirtoon verkossa.
JSON	JavaScript Object Notation, avoimen standardin tiedostomuoto tiedonvälitykseen.

1 Johdanto

Insinöörityön tarkoituksena on luoda yleiskuva progressiivisista verkkosovelluksista sekä tutustua niiden tärkeimpiin ominaisuuksiin ja käytettäviin teknologioihin. Lisäksi työn keskeisenä osana keskitytään progressiivisten verkkosovellusten välimuistin käyttöstrategioihin ja verkottomassa tilassa toimimiseen. Työssä luodaan yksinkertainen prototyppisovellus React-JavaScript-kirjastoa käyttämällä.

Työssä tutustutaan progressiivisten verkkosovellusten käyttämiin teknologioihin ja siihen, mitä niiden avulla voidaan tehdä. Työssä selvitetään, mitä vaaditaan, että verkkosovellusta voidaan kutsua progressiiviseksi verkkosovellukseksi. Progressiivisen verkkosovelluksen todentamiseen tässä insinöörityössä käytetään Googlen kehittämää automatisoitua, avoimeen lähdekoodiin perustuvaa työkalua nimeltä Lighthouse. Lighthouse-työkalun avulla varmistetaan, että sovellus täyttää progressiivisen verkkosovelluksen vaatimukset.

Progressiivisen verkkosovelluksen kehittämiseen on olemassa lukemattomia työkaluja ja ohjelmointikieliä. Tässä insinöörityössä keskitytään niihin, joita käytetään prototyyppisovelluksen kehittämiseen. Lisäksi tutkitaan suosituimpien selainten välisiä valmiuksia hyödyntää progressiivisten verkkosovellusten käyttämiä teknologioita.

Työssä valmistettavan prototyyppisovelluksen avulla tutkitaan erilaisia tapoja käyttää välimuistia ja selvitetään, kuinka välimuistin avulla saadaan luotua hyviä käyttäjäkokemuksia hitaan ja huonon verkkoyhteyden alueella ja mahdollistetaan sovelluksen käyttäminen kokonaan verkottomassa tilassa.

2 Progressiiviset verkkosovellukset

2.1 Määritelmä ja vaatimukset

Progressiiviset verkkosovellukset ovat mobiililaitteille optimoituja verkkosivuja, jotka voidaan asentaa laitteeseen selaimen kautta [1]. Progressiiviset verkkosovellukset mahdollistavat sellaisten toimintojen käytön verkkosovelluksissa, joiden käyttö

aikaisemmin oli mahdollista vain natiiveissa mobiilisovelluksissa. Näin kurotaan umpeen eroja verkkosovellusten ja natiivien mobiilisovellusten välillä.

Jotta verkkosovellusta voidaan kutsua progressiiviseksi verkkosovellukseksi, sen tulee sisältää vähintään kolme asiaa. Sovelluksen pitää rekisteröidä Service worker, jotta se pystyy toimimaan verkottomassa tilassa. Sovelluksen tulee sisältää Web App Manifest, jonka tarjoamia tietoja vaaditaan, että sovellus voidaan asentaa laitteen aloitusnäytölle. Kolmantena vaatimuksena on, että sovellus tarjoillaan HTTPS-protokollaa käyttäen, jotta sen käyttö on turvallista. [2.]

Vaikka teoriassa edellä mainitut kolme kriteeriä riittävät siihen, että verkkosovelluksesta tulee progressiivinen verkkosovellus, se ei vielä käytännössä riitä. Sovelluksen tulee tarjota natiivin mobiilisovelluksen kaltainen käyttäjäkokemus. Se tarkoittaa, että sovelluksen tulee olla responsiivinen ja skaalautua niin mobiililaitteilla kuin työpöytäsovelluksena käytettäessä. Sovelluksen tulee myös toimia ilman selainkomponentteja, kuten osoiterivi ja navigointipainikkeet. Taulukossa 1 esitellään tarkempia vaatimuksia, joiden on täytyttävä, että sovellusta voidaan kutsua progressiiviseksi verkkosovellukseksi. [3.]

Taulukko 1. Vaatimukset, jotka Lighthouse-työkalu on asettanut progressiivisille verkkosovelluksille.

PWA-sovelluksen vaatimukset	
Kriteeri	**Lisätietoa**
Tarpeeksi nopea mobiiliverkoissa	Sovelluksen tulee olla interaktiivinen alle 10 sekunnissa, hitaassakin verkossa.
Sivusto vastaa koodilla 200 verkottomassa tilassa	Sivuston tulee toimia verkottomassa tilassa.
Aloitussivu vastaa koodilla 200 verkottomassa tilassa	Mobiililaitteen aloitussivulle asennetun sovelluksen tulee toimia verkottomassa tilassa.
Käyttää HTTPS-protokollaa	
Uudelleenohjaa HTTP-protokollaa käyttävän liikenteen käyttämään HTTPS-protokollaa	Jos sovellus ei uudelleenohjaa HTTP-protokollaa käyttävää liikennettä käyttämään HTTPS-protokollaa, testi epäonnistuu.
Rekisteröi Service workerin	
Web App Manifest sisältää vaadittavat tiedot	Web App Manifestin tulee sisältää sovelluksen nimi, kuvake, aloitus-web-osoite ja se, kuinka sovellus esitetään. Lisäksi sen

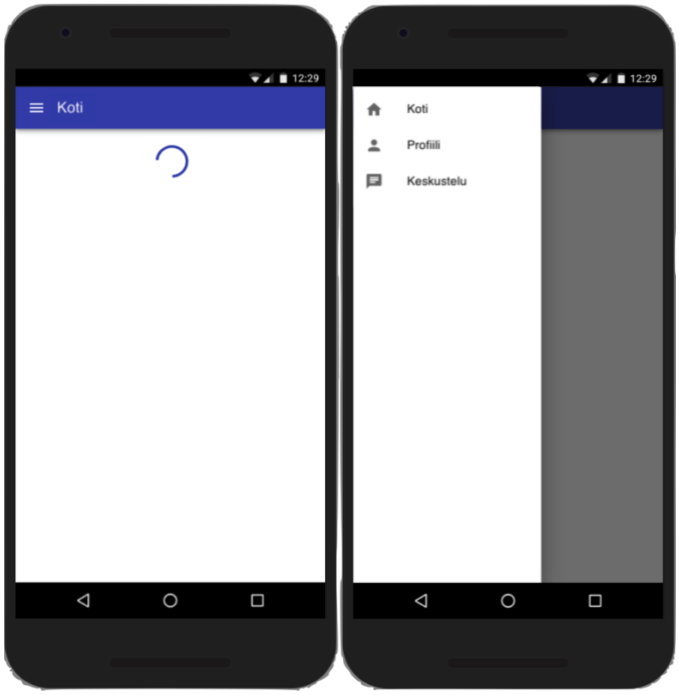
	tulee määrätä, että käytetään kyseistä sovellusta eikä esimerkiksi sovelluksen natiivia versiota.
Käyttää mukautettua aloitusruutua	
Asettaa osoitepalkin värin	Sovelluksen tulee sisältää metatunniste, joka määrittää sovelluksen teemavärin. Myös Web App Manifestin tulee sisältää teemavärin määritys, jotta testi onnistuu.
Näytettävä sisältö skaalautuu ikkunan mukaan	Näytettävän sivun leveys tulee olla yhtä suuri kuin ikkuna jossa se näytetään.
Sisältää sovellusikkunan leveyden määrittävän metatunnisteen	Sovelluksen tulee sisältää viewport-metatunniste, joka sisältää content-määritteen. Content-määritteen tulee sisältää teksti width=.
Näyttää sisältöä, vaikka JavaScript ei olisi käytössä	
Tarjoaa apple-touch-iconin	Sovelluksen tulee sisältää link-tunniste, joka määrittää apple-touch-iconin. Apple-touch-icon on kuvake, jota Apple-laitteet käyttävät sovelluksen kuvakkeena, jos se asennetaan laitteen aloitussivulle.

Kun kaikki kriteerit on täytetty, sovellusta voidaan kutsua progressiiviseksi verkkosovellukseksi. Se on nopea, turvallinen, asennettavissa mobiililaitteen aloitussivulle ja skaalautuu kaikenkokoisille näytöille.

2.2 Sovelluskuori

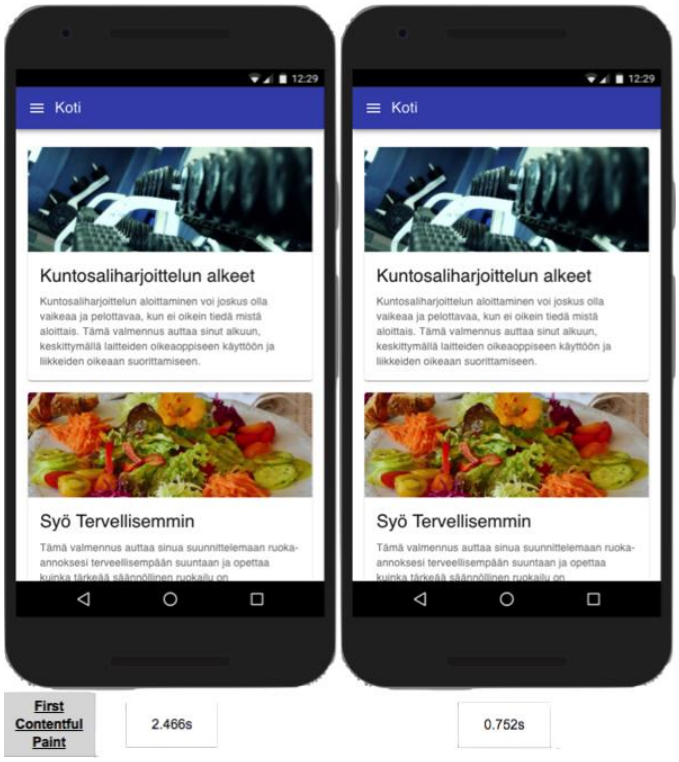
Sovelluskuoriarkkitehtuuri on tapa, jota voidaan käyttää progressiivisten verkkosovellusten luomiseen. Kun sovellus ensimmäisen kerran avataan, se tallentaa välimuistiin minimaalisimmat HTML-, CSS- ja JavaScript-tiedostot, jotka vaaditaan käyttöliittymän näyttämiseen. Tämä mahdollistaa käyttöliittymän välittömän latautumisen seuraavalla kerralla, kun sovellus käynnistetään, sekä sovelluksen toimimisen verkottomassa tilassa. [4.]

Kuvassa 1 on prototyyppisovelluksen sovelluskuori. Se sisältää kolme näkymää, koti-, profiili- ja keskustelusivun. Sovelluskuoreen kuuluu myös navigointivalikko, josta voidaan valita haluttu sivu.



Kuva 1. Prototyyppisovelluksen sovelluskuori.

Sovelluskuori latautuu välittömästi käynnistymisen jälkeen välimuistista. Sovelluskuoren ansiosta käyttäjälle voidaan indikoida, että sovellus on valmis ja sisältö on latautumassa palvelimelta. Kuvassa 2 näkyy prototyyppisovellus, joka on saanut ladattua sisällön palvelimelta. Kuvasta näkyy myös, kuinka suuri ensimmäisen ja toisen latauskerran välinen ero on sovelluskuoren ansiosta. Latausajat on testattu webpagetest.org-sivuston tarjoamalla testillä.



Kuva 2. Prototyyppisovelluksen sovelluskuori, johon on ladattu sisältö palvelimelta.

Palvelimelta ladattava dynaaminen sisältö on mahdollista tallentaa sovelluksen välimuistiin, ja sovelluskuori pystyy käyttämään sitä ensimmäisen latauskerran jälkeen. Käyttäjälle voidaan siis näyttää välittömästi myös sisältöä sovelluskuoren ja välimuistin avulla. Kun uutta sisältöä on saatu palvelimelta, se päivitetään käyttäjän näkyville ja tallennetaan välimuistiin seuraavaa käyttökertaa varten.

2.3 Service worker -teknologia

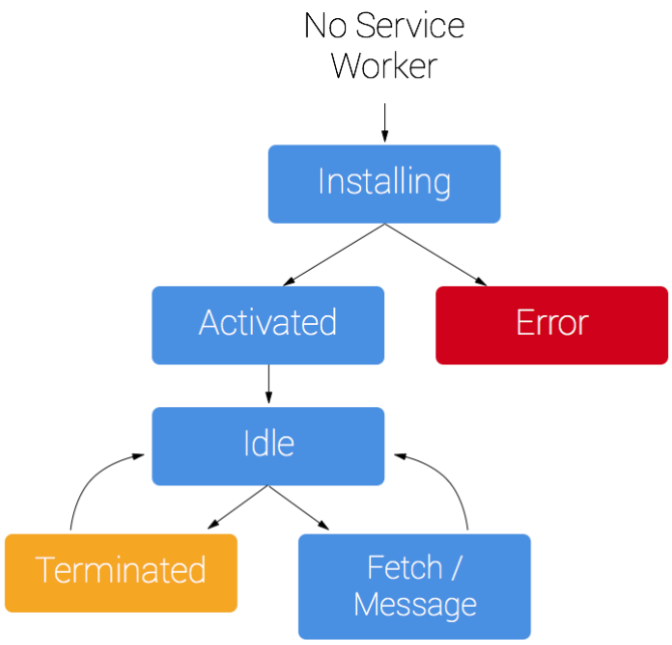
Service worker on yksi progressiivisten verkkosovellusten tärkeimmistä osista. Se on JavaScript-tiedosto, jota suoritetaan omassa säikeessä, erillään selaimen pääsäikeestä [5]. Se mahdollistaa ominaisuuksien, jotka aiemmin kuuluivat vain natiiveihin mobiilisovelluksiin, kuten push-ilmoitusten ja taustasynkronoinnin, implementoimisen verkkosovellukseen [6].

Service workerin avulla pystytään sieppaamaan ja käsittelemään verkkopyyntöjä ja hallitsemaan niistä saatujen vastausten tallentaminen välimuistiin [6].

Service workerin monimutkaisin osuus on niiden elinkaari. Service workerin elinkaaren tarkoitus on varmistaa, että ainoastaan yksi versio sivustosta tai sovelluksesta on käynnissä kerrallaan. Elinkaaren ansiosta Service worker pystyy päivityksiä havaitessaan luomaan uuden version itsestään häiritsemättä nykyistä. Kun vanhaa versiota käyttävä sivusto tai sovellus suljetaan, Service workerin vanha versio korvataan uudella ja otetaan käyttöön, kun sivusto tai sovellus avataan seuraavan kerran. Elinkaaren rakenne mahdollistaa myös yhteydetöntila ensin -strategian käytön. [7.]

Se, että sivustosta tai sovelluksesta on ainoastaan yksi versio käynnissä kerrallaan, on hyvin tärkeää. Jos samasta sivustosta tai sovelluksesta on käynnissä useampi eri versio samanaikaisesti, se voi johtaa virheilmoituksiin ja tiedon katoamiseen. [7.]

Kuvassa 3 on esitetty yksinkertaistettu Service workerin elinkaari. Ensimmäiseksi Service worker pitää rekisteröidä. Rekisteröinnin yhteydessä tulee varmistaa, että selain tukee Service workerin käyttöä. Jos selain tukee Service workerin käyttöä ja se saadaan rekisteröityä, alkaa seuraava elinkaaren vaihe.



Kuva 3. Service workerin elinkaari yksinkertaistettuna [6].

Elämänkaaren seuraava vaihe on Service workerin asennus. Service worker saadaan asennettua, jos sitä ei ole aikaisemmin rekisteröity tai sen skripti on muuttunut edes

yhden tavun verran. Jos kumpikaan näistä ei ole totta, Service worker ei asennu ja vikailmoitustapahtuma laukeaa. [8.]

Onnistuneen asennuksen jälkeen aktivointitapahtuma käynnistyy. Aktivointi onnistuu, jos asennustapahtuman yhteydessä on annettu käsky ohittaa odottaminen, asennushetkellä ei ole aikaisempaa aktiivista Service workeriä tai käyttäjä päivittää sivun. Jos asennus ei onnistu, Service worker odottaa, kunnes asentaminen on mahdollista. [8.]

Asentumisen valmistuttua Service workerillä on kontrolli sivustosta. Tämän jälkeen se menee valmiustilaan odottamaan selaimen tuottamien tapahtumien vastaanottamista. Tällaisia tapahtumia ovat esimerkiksi verkkopyynnöt ja sivustolta Service workerille lähetetyt viestit. Jos Service worker ei hetkeen vastaanota tapahtumia, se menee lopetettuun tilaan, josta se palaa takaisin valmiustilaan, kun jokin tapahtuma vastaanotetaan. [8.]

Ensimmäinen selain, joka tuki kokonaisuudessaan Service workerin käyttöä, oli Google Chrome. Se saavutti täydellisen tuen Service workerin käyttöön vuoden 2015 loppupuolella. Mozilla Firefox sai implementoitua Service workerin käytön kokonaisuudessaan heti 2016 vuoden alussa. Suosituimpien selainten joukosta, viimeisenä Service workerin implementoinnin valmiiksi sai Safari, vasta vuoden 2018 loppupuolella [9].

2.4 Web App Manifest -tiedosto

Web App Manifest on JSON-tiedosto, joka antaa selaimelle tietoa sovelluksesta. Se on yksi osa, joka vaaditaan, että tavallisesta verkkosovelluksesta voidaan tehdä progressiivinen verkkosovellus. Web App Manifest mahdollistaa progressiivisen verkkosovelluksen asentamisen mobiililaitteen aloitussivulle ja näin ollen luo natiivin mobiilisovelluksen kaltaisen vaikutelman. [10.]

Jotta progressiivisen verkkosovelluksen pystyy lataamaan mobiililaitteen aloitusnäytölle, tulee Web App Manifestin sisältää tietyt asiat ja täyttää minimaaliset vaatimukset. Kuten esimerkkikoodissa 1 on esitetty, Web App Manifestissa tulee määritellä sovelluksen nimi, kuvake, aloitus-web-osoite ja se, kuinka sovellus esitetään. [11.]

```
{
  "short_name": "Oppari",
  "name": "Opinnäytetyö",
  "icons": [
    {
      "src": "logo192.png",
      "type": "image/png",
      "sizes": "192x192"
    },
    {
      "src": "logo512.png",
      "type": "image/png",
      "sizes": "512x512"
    }
  ],
  "start_url": ".",
  "display": "standalone",
}
```

Esimerkkikoodi 1. Web App Manifestin minimivaatimukset.

Web App Manifest ei ole vielä tuettu tai se on tuettu vain osittain monessa selaimessa. Taulukossa 2 esitetään suosituimpien selainten väliset ja laitekohtaiset erot käyttää Web App Manifestia.

Taulukko 2. Web App Manifestin käytön tuki eri selaimissa [12].

	Web App Manifest		
	Työpöytäselain	**iOS**	**Android**
Chrome	Tuettu	Osittain tuettu	Tuettu
Safari	Ei tuettu	Osittain tuettu	-
Firefox	Ei tuettu	-	Osittain tuettu

Google Chrome on jälleen edelläkävijän paikalla: se on saanut implementoitua Web App Manifestin käytön jo kokonaisuudessa. Mozilla Firefox ja Safari tukevat sen käyttöä osittain. [12.]

2.5 Taustasynkronointi

Taustasynkronoinnin avulla progressiivinen verkkosovellus pystyy varmistamaan, että käyttäjän lähettämät pyynnöt oikeasti lähtevät eteenpäin. Se myös mahdollistaa sovelluksen käyttämisen, vaikka pyyntö ei olisi vielä lähtenytkään eteenpäin. [13.]

Taustasynkronointi toimii niin, että samalla kun sovelluksesta lähetetään verkkopyyntö**,** rekisteröidään synkronointioperaatio**.** Verkkopyyntö kaapataan Service workerin avulla ja pyritään lähettämään eteenpäin. Jos pyyntö onnistuu, käsitellään saatu vastaus eikä muita operaatioita tarvitse tehdä. Jos pyyntö epäonnistuu, se tallennetaan selaimen tietokantaan ja vastuu pyynnön lähettämisestä siirtyy taustasynkronointioperaatiolle. Operaatio pyrkii lähettämään tietokantaan tallennettuja pyyntöjä, kunnes onnistuu siinä. Kun pyyntö on onnistunut, se poistetaan tietokannasta ja käsitellään saatu vastaus.

Taulukossa 3 esitetään suosituimpien selainten väliset ja laitekohtaiset erot käyttää taustasynkronointia. Se on vielä uusi teknologia ja siksi hyvin heikosti tuettuna eri selaimissa.

Taulukko 3. Taustasynkronoinin käytön tuki eri selaimissa [14].

Taustasynkronointi

	Työpöytäselain	**iOS**	**Android**
Chrome	Tuettu	Ei tuettu	Tuettu
Safari	Ei tuettu	Ei tuettu	-
Firefox	Ei tuettu	-	Ei tuettu

Suosituimmista selaimista Google Chrome on ainoa joka tukee taustasynkronoinnin käyttöä. Mozilla Firefox tai Safari eivät kumpikaan vielä tue taustasynkronoinnin käyttöä edes osittain. [14.]

2.6 Push-ilmoitukset

Push-ilmoitukset ovat viestejä, jotka näytetään käyttäjän laitteen ilmoituksissa eikä sovelluksen sisällä. Ne rakentuvat kahdesta eri rajapinnasta, Notifications API ja Push API. Notifications API hoitaa ilmoituksen näyttämisen käyttäjälle, ja Push API mahdollistaa Service workerin kontrolloida ilmoituksia, jotka saapuvat palvelimelta, vaikka sovellus ei olisi avoinna [15].

Push-ilmoituksia käyttämällä pyritään saamaan käyttäjät avaamaan sovellus ja käyttämään sitä enemmän. Niiden avulla on mahdollista saada käyttäjät palaamaan sovellukseen, joka on mahdollisesti päässyt unohtumaan. Push-ilmoituksia on myös

mahdollista personoida käyttäjien profiilin mukaan, esimerkiksi jos tahdotaan mainostaa kampanjaa tietylle kohderyhmälle. [16.]

Push-ilmoituksia on kahta eri tyyppiä, paikallinen ja palvelimelta lähetetty. Paikallinen ilmoitus voidaan lähettää sovelluksesta sen ollessa avoinna, ja myös taustalla avoinna olevasta sovelluksesta voidaan lähettää ilmoitus. Palvelimelta lähetetty ilmoitus ei vaadi, että sovellus on avoinna [17]. Taulukosta 4 selviävät suosituimpien selainten väliset ja laitekohtaiset erot paikallisten push-ilmoitusten käytön tuesta, ja taulukosta 5 selviää palvelimelta lähetettyjen push-ilmoitusten tuki.

Taulukko 4. Paikallisten push-ilmoitusten käytön tuki eri selaimissa [18].

Paikalliset push-ilmoitukset

	Työpöytäselain	**iOS**	**Android**
Chrome	Tuettu	Ei tuettu	Tuettu
Safari	Tuettu	Ei tuettu	-
Firefox	Tuettu	-	Tuettu

Notification API:a, joka mahdollistaa paikallisten ilmoitusten näyttämisen käyttäjälle, tuetaan jo kaikissa yleisimmin käytetyiden selainten työpöytäversioissa. iOS-käyttöjärjestelmä ei tue vielä Notification API:n käyttöä, joten mobiililaitteilla, jotka käyttävät iOS-käyttöjärjestelmää, ei voida näyttää push-ilmoituksia.

Taulukko 5. Palvelimelta lähetettyjen push-ilmoitusten käytön tuki eri selaimissa [19].

Palvelimelta lähetetyt push-ilmoitukset

	Työpöytäselain	**iOS**	**Android**
Chrome	Tuettu	Ei tuettu	Tuettu
Safari	Ei tuettu	Ei tuettu	-
Firefox	Tuettu	-	Tuettu

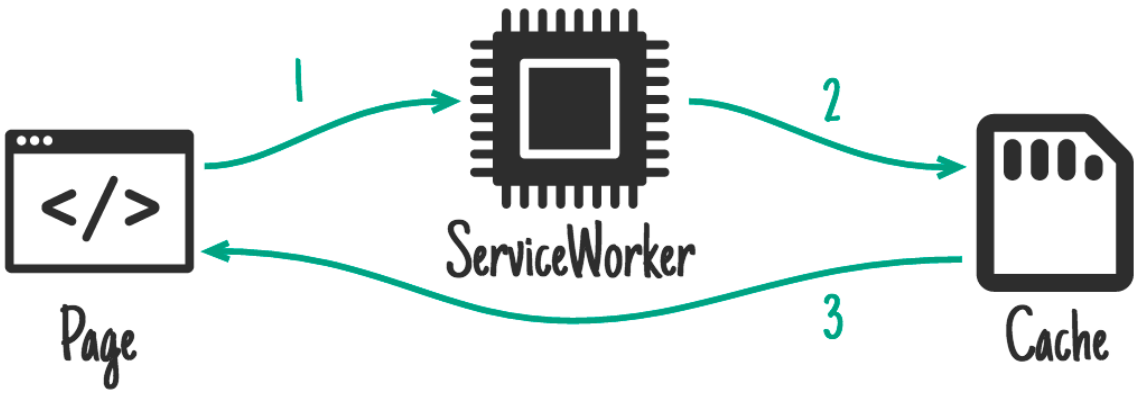
Jotta progressiivinen verkkosovellus pystyy vastaanottamaan palvelimelta lähetettyjä push-ilmoituksia, tulee käytetyn selaimen tukea Push API:n käyttöä. Tämä on uusi teknologia ja siksi vielä heikosti tuettuna selaimissa.

3 Välimuistin käyttöstrategiat

Yleisimmin käytettyjen välimuistin käyttöstrategioiden käyttöön ja toteutukseen löytyy paljon apua ja esimerkkejä verkosta. Esiteltäviä strategioita apuna käyttäen progressiiviset verkkosovellukset saadaan toimimaan tehokkaammin ja latautumaan nopeammin hyvinkin hitaan verkon alueella tai jopa kokonaan verkottomassa tilassa

3.1 Vain välimuisti -strategia

Vain välimuisti -strategia tarkoittaa, että sovellus sallii pyynnön vastauksen tarjoiltavan vain välimuistista. Tämän strategian käyttö onnistuu ainoastaan, jos jokin toinen prosessi on tallentanut välimuistiin käytettävät resurssit, ennen kuin sovellus pyytää niitä [20]. Kuvassa 4 on havainnollistettu vain välimuisti -strategian tietovirran kulun vaiheet. Tietovirta tarkoittaa sivustolta lähetettävän tiedon ja siihen saadun vastauksen kulkua sovelluksessa.

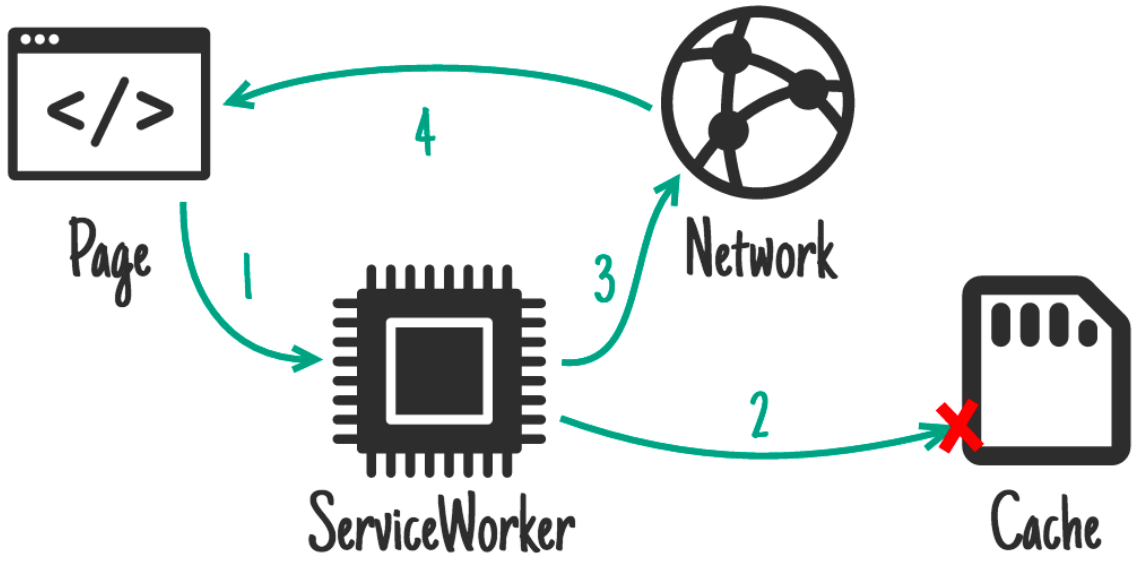


Kuva 4. Vain välimuisti -strategian käyttö visualisoituna [21].

Tätä strategiaan käytetään sovelluksen staattisen sisällön, joka ei koskaan muutu, tarjoamiseen [22]. Käyttökohde progressiivisessa verkkosovelluksessa voi olla esimerkiksi sovelluskuori.

3.2 Välimuisti, varalla verkko -strategia

Välimuisti, varalla verkko -strategia pyrkii hakemaan vastauksen pyyntöön ensin välimuistista. Jos välimuistissa ei ole vielä tallennettuja resursseja, ne pyritään hakemaan verkosta. Jos verkkopyyntö onnistuu, tarjoillaan käyttäjälle saatu vastaus ja tallennetaan se välimuistiin seuraavaa kertaa varten. [23.] Kuva 5 havainnollistaa, kuinka tietovirta etenee sovelluksessa tätä strategiaa käytettäessä.



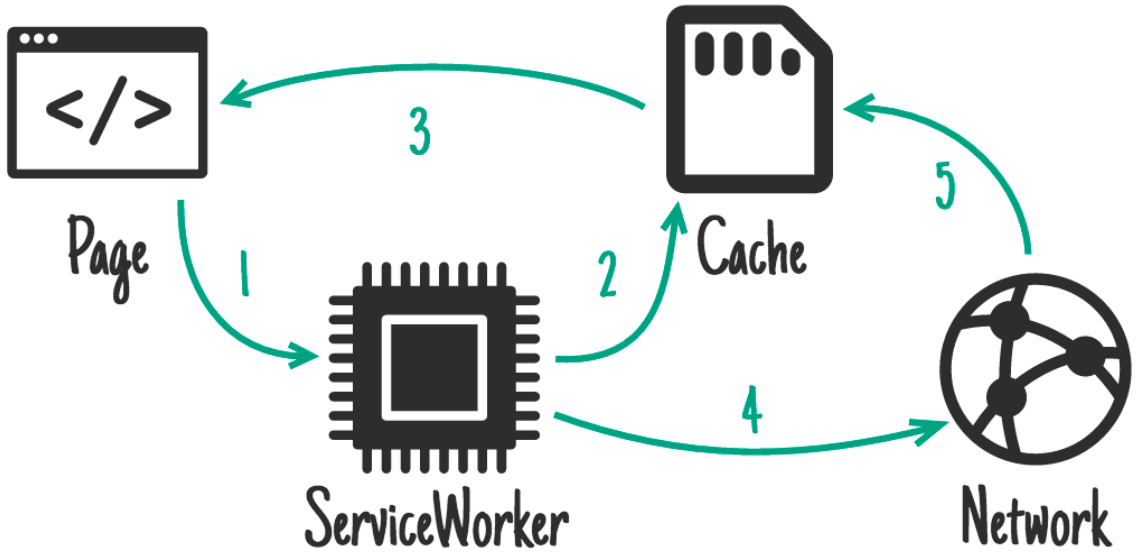
Kuva 5. Välimuisti, varalla verkko -strategian käyttö visualisoituna [21].

Tätä strategiaa voidaan käyttää sovelluksen resurssien ja toistuvien, samoihin resursseihin kohdistuvien pyyntöjen välittömään tarjoiluun [23]. Tällaisia resursseja voivat olla esimerkiksi staattiset kuvat ja CSS-tiedostot, joita sovellus käyttää.

3.3 Välimuisti varmistaa datan käyttökelpoisuuden -strategia

Välimuisti varmistaa datan käyttökelpoisuuden -strategiaa käytettäessä tarkistetaan ennen verkkopyynnön lähettämistä, onko välimuistiin tallennettu dataa, jota voitaisiin tarjoilla käyttäjälle. Jos välimuistissa ei ole aikaisempaa dataa, joutuu käyttäjä odottamaan sen latautumista verkosta. Jos välimuistista löytyy aikaisemmin tallennettua dataa, tarkistetaan, onko se vanhentunutta. Jos data ei ole enää käyttökelpoista, sitä ei tarjoilla käyttäjälle vaan käyttäjä joutuu odottamaan verkkopyynnön valmistumista ja

sieltä saatua dataa. Jos välimuistissa oleva data on käyttökelpoista, se tarjoillaan käyttäjälle välittömästi, mutta sovellus lähettää silti verkkopyynnön. Kun verkkopyynnöstä on saatu vastaus, se tallennetaan välimuistiin ja tarjoillaan käyttäjälle seuraavalla kerralla, kun hän tekee kyseisen verkkopyynnön [24]. Kuva 6 auttaa ymmärtämään, kuinka datavirta kulkee läpi sovelluksen, jos käytetään tätä strategiaa.



Kuva 6. Välimuisti varmistaa datan käyttökelpoisuuden -strategian käyttö visualisoituna [21].

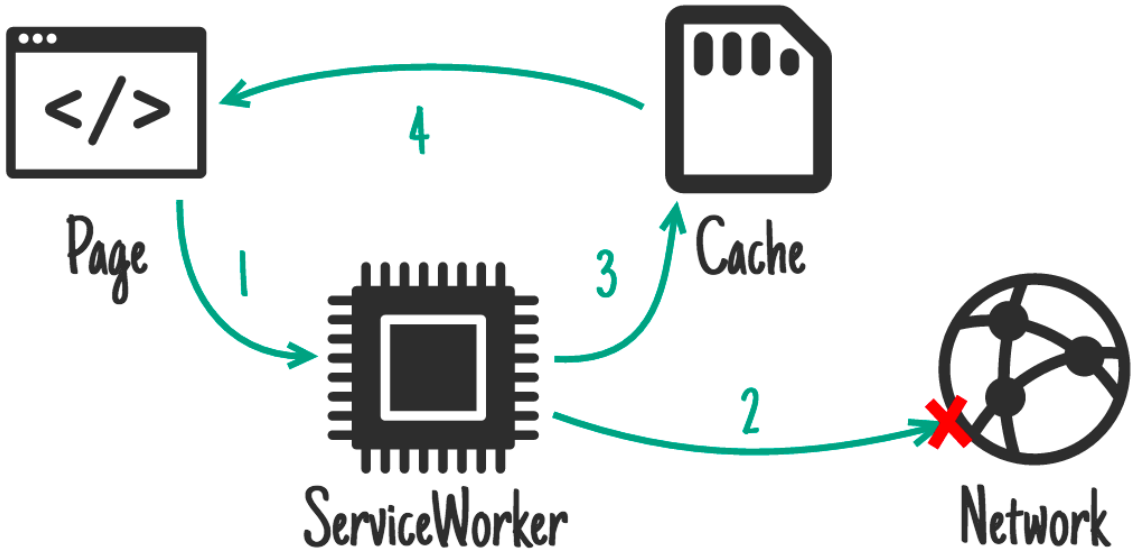
Tätä strategiaa voidaan käyttää tilanteissa, joissa ei ole kriittistä, että data on uusinta mahdollista mutta tahdotaan kuitenkin rajoittaa välimuistista haettavan datan ikää. Tällainen käyttökohde voi olla esimerkiksi uutisotsikot. [25.]

Käyttäjät monesti hakevat uutisia useamman kerran päivässä. Uutisotsikot eivät kuitenkaan päivity jatkuvasti, joten välimuistin datalle voidaan asettaa maksimi-iäksi esimerkiksi kaksi tuntia. Jos käyttäjä ei ole katsonut uutisia yli kahteen tuntiin, hänelle ei enää näytetä vanhoja uutisia välimuistista. Mutta jos edellisestä kerrasta on alle kaksi tuntia, käyttäjä saa vielä vanhat uutiset, mutta välimuistiin päivitetään uusimmat ja tarjoillaan seuraavalla kerralla.

3.4 Verkko, varalla välimuisti -strategia

Joissain tapauksissa on tärkeää, että käytettävissä on aina uusin mahdollinen data. Tällaisia tapauksia voivat olla esimerkiksi sosiaalisen median kommentti- ja

keskusteluosiot. Keskusteltaessa toisen henkilön kanssa on tärkeää, että mahdolliset uudet viestit päivittyvät käyttäjälle välittömästi. Tietovirran kulku sovelluksessa tätä strategiaa käytettäessä on havainnollistettu kuvassa 7.



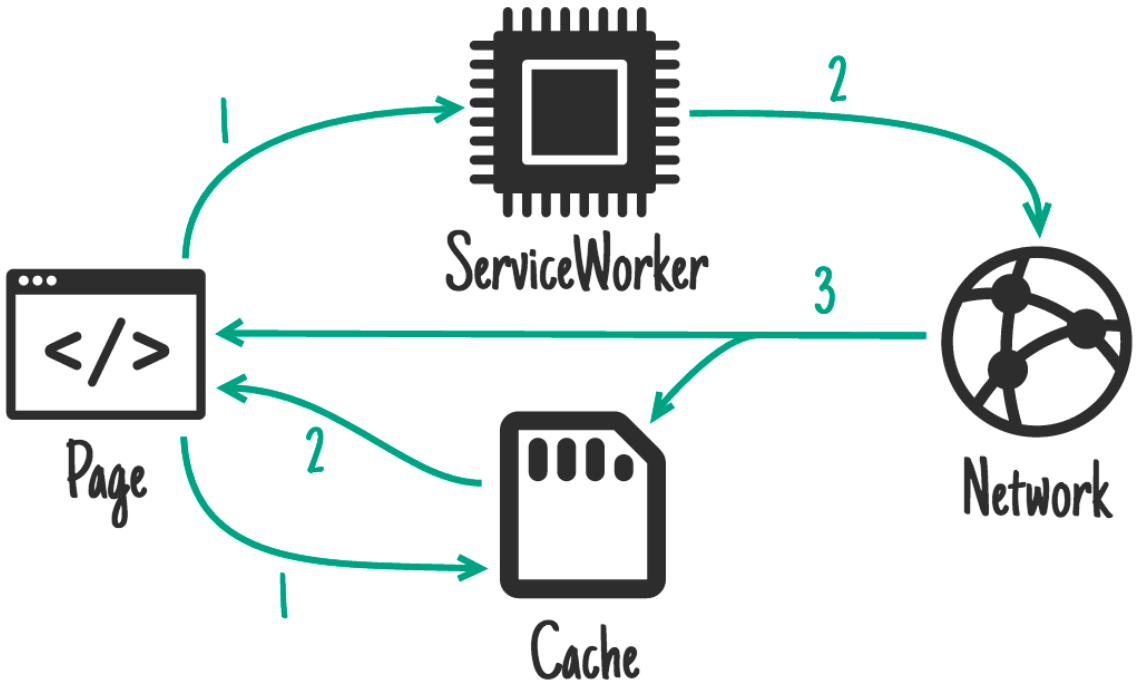
Kuva 7. Verkko, varalla välimuisti -strategian käyttö visualisoituna [21].

Tällaisessa tilanteessa voidaan käyttää verkko, varalla välimuisti -strategiaa. Se tarkoittaa sitä, että sovellus yrittää aina ensin ladata uuden tiedon palvelimelta, mutta jos se ei onnistu, turvaudutaan välimuistissa olevaan vanhaan tietoon. Tässä strategiassa on kuitenkin se huono puoli, että jos käyttäjällä on hidas verkkoyhteys tai vastausta palvelimelta ei saada, käyttäjä joutuu odottamaan verkkopyynnön epäonnistumisen, ennen kuin saa näkyville vanhoja viestejä välimuistista. [21.]

3.5 Ensin välimuisti, sitten verkko -strategia

Ensin välimuisti, sitten verkko -strategia on melko samankaltainen kuin verkko turvautuu välimuistiin -strategia. Se on tarkoitettu myös sellaisen sisällön tarjoamiseen käyttäjälle jonka tulee olla aina mahdollisimman uutta. Tässä strategiassa käyttäjälle tarjotaan välittömästi välimuistiin tallennettu sisältö ja taustalla lähetetään verkkopyyntö, joka pyrkii noutamaan palvelimelta mahdollisesti päivittyneen sisällön. Jos palvelimelta saadaan uutta sisältöä, päivitetään se käyttäjän näkyville. Tässä strategiassa on kuitenkin tärkeä indikoida käyttäjälle, että uutta sisältöä haetaan palvelimelta, ettei käyttäjä saa käsitystä, että uutta sisältöä ei ole.

Tässä tapauksessa luodaan pelkän verkkopyynnön sijaan myös pyyntö välimuistiin. Käyttäjä saa välittömästi näkyviin välimuistissa olevan sisällön, verkkopyyntö kaapataan Service workerin avulla ja kontrolloidaan saatu vastaus. Jos palvelimelta saadaan uutta dataa, se tallennetaan välimuistiin ja päivitetään käyttäjän näkyville verkkopyynnön valmistuttua. Ensin välimuisti, sitten verkko -strategiaa käytettäessä sovelluksen tietovirta kulkee kuvan 8 mukaisesti.



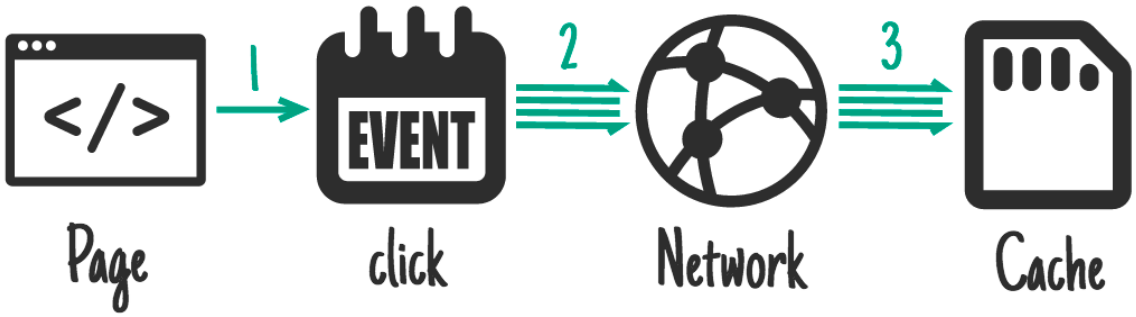
Kuva 8. Ensin välimuisti, sitten verkko -strategian käyttö visualisoituna [21].

Joissain tapauksissa vanha sisältö voidaan vain korvata uudella. Mutta tämä ei aina ole hyvä käytäntö: jos käyttäjä on esimerkiksi lukemassa sosiaalisen median päivitystä, joka on latautunut hänelle välimustista, ei sisältöä voi vain heittää pois käyttäjän näkyviltä kesken lukemisen ja antaa tilalle uutta sisältöä. Uusi sisältö kannattaa esimerkiksi lisätä listaan ja indikoida käyttäjälle, että sitä on, jolloin käyttäjä pystyy lukemaan vanhan sisällön loppuun ja sen jälkeen selaamaan uutta. [21.]

3.6 Käyttäjä kontrolloi välimuistia -strategia

Käyttäjä kontrolloi välimuistia -strategiassa käyttäjällä on täysi kontrolli siihen, mitä välimuistiin tallennetaan. Sovellukseen tulee laittaa nappi, josta käyttäjä valitsee, että

tahtoo tallentaa kyseisen resurssin. Service worker kaappaa tapahtuman, noutaa resurssit verkosta ja tallentaa ne välimuistiin [21]. Tätä strategiaa käytettäessä tietovirran eteneminen ja pyyntöjen luonti sovelluksessa on esitetty kuvassa 9.



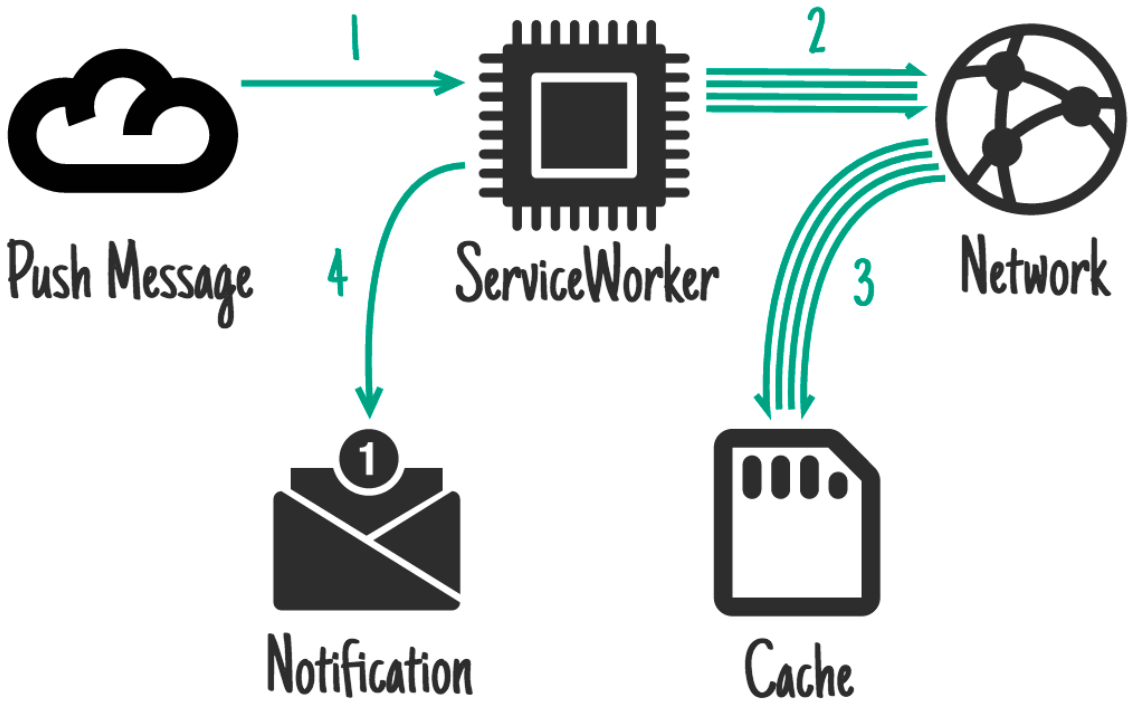
Kuva 9. Käyttäjä kontrolloi välimuistia -strategian käyttö visualisoituna [21].

Käyttäjä kontrolloi välimuistia -strategiaa käytetään tilanteessa, jossa kaikkien resurssien tallentamineen välimuistiin ei kannatta. Tällainen tilanne voi olla esimerkiksi, jos sovellukseen ladataan useita videoita. Kaikkia videoita ei voida tallentaa välimuistiin rajoitetun tilan takia, joten käyttäjä voi valita itselleen tärkeimmät videot. Toinen mahdollinen tilanne, jossa tätä strategiaa voi käyttää, on nopeasti poistuvien resurssien tallentaminen. Uutisartikkelit ovat hyvä esimerkki: jos käyttäjä ei ehdi lukea artikkelia, hän voi tallentaa sen välimuistiin ja lukea myöhemmin. Näin hänen ei tarvitse pelätä, että artikkeli poistuu, ennen kuin hän on sen lukenut.

3.7 Push-ilmoituksen yhteydessä -strategia

Safari-selain ja iOS-käyttöjärjestelmä eivät tue vielä push-ilmoitusten käyttöä progressiivisissa verkkosovelluksissa [19], joten tämä strategia koskee enemmän Chrome-selaimella tai Android-laitteilla käytettäviä sovelluksia.

Strategian toimintaperiaate on hyvin yksinkertainen: kun sovellus vastaanottaa push-ilmoituksen, Service worker käynnistyy ja noutaa vaadittavan datan. Tämän jälkeen noudettu data tallennetaan välimuistiin. Pyyntöjen luonti ja tietovirran kulku sovelluksessa tapahtuvat kuvan 10 havainnollistamalla tavalla tätä strategiaa käytettäessä.



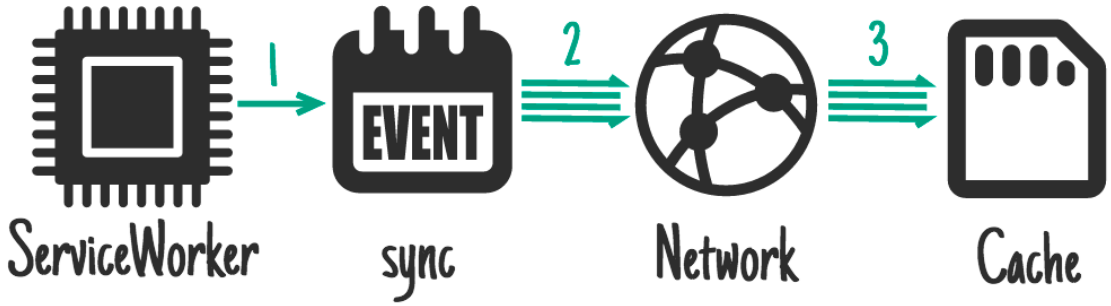
Kuva 10. Push-ilmoituksen yhteydessä -strategian käyttö visualisoituna [21].

Tilanne, jossa push-ilmoituksen yhteydessä -strategiaa käytetään, on kun vaaditaan tiedon välitöntä synkronointia, kuten tehtävälistan tai kalenteritapahtumien päivittyminen. Toinen mahdollinen tilanne on push-ilmoitukseen liittyvän datan, kuten uuden viestin tai sähköpostin, tallentaminen välimuistiin. [21.]

3.8 Taustasynkronoinnin yhteydessä -strategia

Taustasynkronoinnin yhteydessä **-**strategian käyttöä rajoittaa selainten tuki käyttää taustasynkronointia. Yleisimmin käytettyjen selainten joukosta ainoastaan Chrome tukee taustasynkronoinnin käyttöä. [14.]

Tätä strategiaa voidaan käyttää tilanteissa, joissa välimuistin päivittäminen ei ole kiireellistä. Tällaisia voivat olla esimerkiksi uutisartikkelit ja sosiaalisen median syötteet [14]. Kuva 11 havainnollistaa pyyntöjen luonnin ja tietovirran kulun sovelluksessa tätä strategiaa käytettäessä.



Kuva 11. Taustasynkronoinnin yhteydessä -strategian käyttö visualisoituna [21].

Taustasynkronoinnin yhteydessä -strategiassa välimuisti päivitetään, kun jokin tapahtuma on saatu suoritettua onnistuneesti. Sovelluksesta voidaan lähettää pyyntö, jolla pyritään noutamaan tietoa. Vaikka pyynnön lähettäminen ei onnistuisi, sovellus voidaan sulkea ja taustasynkronointi yrittää lähettää pyyntöä, kunnes se onnistuu. Kun pyyntö on onnistunut, vastauksena saatu data tallennetaan välimuistiin. [21.]

3.9 Muiden kuin GET-verkkopyyntöjen kontrollointi

Muita kuin GET-verkkopyyntöjä ei pysty tallentamaan välimuistiin. Selaimen tietokantaa ja taustasynkronointia apuna käyttäen on kuitenkin mahdollista luoda käyttäjälle kokemus, jossa hän voi tehdä muitakin verkkopyyntöjä verkottomassa tilassa. Nämä pyynnöt eivät kuitenkaan todellisuudessa lähde eteenpäin eikä niihin saada vastausta, ennen kuin käyttäjä on saavuttanut verkkoyhteyden.

4 Prototyyppisovellus

Insinöörityössä kehitetyn prototyyppisovelluksen tarkoituksena on havainnollistaa, kuinka sovellus saadaan toimimaan myös verkottomassa tilassa erilaisia välimuistin käyttöstrategioita käyttämällä.

Prototyyppisovellus kehitetiin React-Javascript-kirjastoa käyttämällä ja käyttöliittymän luomisessa käytettiin apuna Material-UI-viitekehystä. Sovellus luodaan Create React App -menetelmällä, joka on virallinen tapa luoda uusi React single-page -sovellus, ja se tarjoaa valmiiksi sisäänrakennetut kokoamisasetukset ilman erillisiä konfiguraatioita.

Luotu sovellus sisältää valmiiksi kaikki tarvittavat elementit progressiivisen verkkosovelluksen luomiseksi.

4.1 Käytetyt teknologiat

Prototyyppisovelluksen kehittämiseen valittiin React. React on Facebookin kehittämä, avoimeen lähdekoodiin perustuva JavaScript-kirjasto käyttöliittymien luomiseen. Sen kehittämistä johtaa Facebookille työskentelevä pieni ryhmä, mutta suuri osa kehityksestä tulee ulkopuolisilta yrityksiltä, yhteisöiltä ja yksilöitä. [26.]

React valittiin prototyyppisovelluksen kehittämiseen, koska se tarjoaa Service workerin valmiiksi sisällytettynä projektiin. Valmiiksi implementoitu Service worker on kuitenkin melko rajoitettu, ja sen muokkaaminen vaatii kiertotien käyttöä. Tästä on tehty useita muokkauspyyntöjä Reactin kehittäjille, mutta vielä he eivät ole pyydettyjä muokkauksia tehneet. [27.]

Service workerin muokkaamiseen React-sovelluksessa löytyy kuitenkin tapoja, joista voi valita itseään miellyttävimmän. Tässä opinnäytetyössä päädyttiin tapaan, jossa varsinaisesti alkuperäistä Service workeriä ei muokata, vaan siihen ainoastaan kirjoitetaan lisää. Tämä onnistuu luomalla uusi JavaScript-pohjainen tiedosto, jota tässä työssä kutsutaan nimellä sw-epilog.js. Luotuun tiedostoon kirjoitetaan koodi, joka halutaan lisätä Service workeriin. Tämän jälkeen pitää luoda skripti (esimerkkikoodi 2), joka kertoo sovellukselle, että sw-epilog.js-tiedostossa oleva koodi lisätään varsinaiseen Service workeriin. Kun sovellus kootaan, skripti ajetaan ja saadaan käyttöön halutut lisäykset Service workeriin.

```
"scripts": {
    "sw-epilog": "cat src/sw-epilog.js >> build/service-worker.js",
    "build": "react-scripts build && npm run sw-epilog",
  }
```

Esimerkkikoodi 2. Prototyyppisovelluksen kokoamisskripti.

Prototyyppisovelluksen kehittämiseen Reactin tukena käytetään Material-UI-viitekehystä. Se on joukkorahoituksella toimiva, avoimeen lähdekoodiin perustuva projekti. Sitä ylläpitää pieni ydinryhmä, mutta sen kehittämiseen osallistuu suuri yhteisö.

Se tarjoaa valmiita ja helposti implementoitavia React-komponentteja nopeuttamaan ja helpottamaan verkkosovellusten kehittämistä. [28.]

Material-UI-viitekehystä päätettiin käyttää prototyyppisovelluksen käyttöliittymän luomiseen, koska sen avulla pystytään nopeuttamaan sovelluksen kehittämistä. Materail-UI-viitekehys tarjoaa todella laajan valikoiman helposti muokattavissa olevia valmiita komponentteja React-sovelluksiin.

Prototyyppisovelluksen pakettien hallintaan käytetään npm-paketinhallintajärjestelmää. Npm-paketinhallintajärjestelmä on tarkoitettu JavaScript-kielellä kirjoitettujen pakettien asentamiseen ja jakamiseen.

4.2 Välimuistin käyttö

Insinöörityön prototyyppisovellus sisältää kolme eri näkymää, pääsivun, keskustelusivun ja profiilisivun. Hyvin erilaisten käyttötarkoitusten ansiosta näitä kolmea sivua käyttämällä pystytään havainnollistamaan erilaisia välimuistin käyttöstrategioita.

Kotisivu

Prototyyppisovelluksen kotisivulle haetaan tarjolla olevat valmennukset. Koska se on ensimmäinen näkymä jonka käyttäjä näkee, on tärkeää että näkyville saadaan sisältöä mahdollisimman nopeasti. Jos sovelluksen latautuminen kestää liian kauan, käyttäjät kyllästyvät ja lopettavat sovelluksen käytön eivätkä palaa takaisin.

Tällaisessa tapauksessa paras mahdollinen strategia, jota käyttää, on ensin välimuisti, sitten verkko. Tämän strategian avulla käyttäjälle saadaan välittömästi sisältöä luettavaksi, mutta kuitenkin uusin mahdollinen data päivittyy käyttäjälle heti kun mahdollista.

Itse sovelluksen puolella luodaan kaksi pyyntöä (esimerkkikoodi 3), toinen välimuistiin ja toinen palvelimelle. Välimuistista haetut valmennukset tarjoillaan välittömästi käyttäjälle. Kun verkkopyyntö on valmistunut, päivitetään palvelimelta saatu data käyttäjälle.

```
getRequests = async () => {
    var networkDataReceived = false;

    this.setState({ isLoading: true })
    const url = 'https:// back-opinnaytetyo.herokuapp.com/api/v1/valmennukset'

    var networkUpdate = fetch(url).then((response) => {
        return response.json();
    }).then((data) => {
        networkDataReceived = true;
        this.setState({ data: data })
    });

    // fetch cached data
    caches.match(url).then((response) => {
        if (!response) throw Error("No data");
        return response.json();
    }).then((data) => {
        if (!networkDataReceived) {
            this.setState({ data: data })
        }
    }).catch(() => {
        return networkUpdate;
    }).catch((e) => console.log(e)).then(() => {
        this.setState({ isLoading: false });
    });
}
```

Esimerkkikoodi 3. Ensin välimuisti, sitten verkko -strategiassa luotavat pyynnöt.

Service workerin avulla verkkopyyntö kaapataan (esimerkkikoodi 4) ja kontrolloidaan saadun vastauksen tarjoaminen käyttäjälle. Kun vastaus verkkopyyntöön on saatu, avataan välimuisti ja tallennetaan saatu vastaus sinne. Kun vastaus on tallennettu välimuistiin, se tarjoillaan käyttäjälle.

```
self.addEventListener('fetch', (event) => {
    event.respondWith(
        caches.open('mysite-dynamic').then(async (cache) => {
            const response = await fetch(event.request);
            cache.put(event.request, response.clone());
            return response;
        })
    );
});
```

Esimerkkikoodi 4. Ensin välimuisti, sitten verkko -strategian koodi, jota Service worker käyttää kontrolloidakseen verkkopyyntöjä.

Keskustelusivu

Keskustelusivulle haetaan käyttäjän ja valmentajan väliset viestit. Tällaisessa tilanteessa tahdotaan, että käyttäjällä on aina käytössään kaikkein uusimmat viestit, joten voidaan valita kahden eri strategian välillä. Voidaan käyttää strategiaa, joka hakee viestit

palvelimelta, mutta jos se ei onnistu, käytetään välimuistiin tallennettuja viestejä (esimerkkikoodi 5). Toinen vaihtoehto on, että näytetään käyttäjälle välittömästi viestit välimuistista, mutta samalla haetaan uusia viestejä palvelimelta. Jos palvelimelta saadaan uusia viestejä, ne päivitetään käyttäjän näkyville.

```
self.addEventListener('fetch', (event) => {
    event.respondWith(
        fetch(event.request).then((response) => {
            caches.open('mysite-dynamic').then(async (cache) => {
                cache.put(event.request, response)
            });
            return response.clone();
        }).catch(() => {
            return caches.match(event.request);
        })
    );
});
```

Esimerkkikoodi 5. Verkko, varalla välimuisti -strategian koodi, jota Service worker käyttää kontrolloidakseen verkkopyyntöjä.

Prototyyppisovelluksen keskustelusivun osalta päädyttiin käyttämään verkko turvautuu välimuistiin -strategiaa. Näin saadaan varmistettua, että käyttäjällä on aina saatavilla uusimmat mahdolliset viestit. Tämän strategian ansiosta myös käyttäjä tietää, että hänelle on varmasti kaikki uusimmat viestit, koska hän on joutunut katsomaan, kun niitä ladataan.

Profiilisivu

Prototyyppisovelluksen profiilisivulla näytetään käyttäjän henkilökohtaisia tietoja, kuten nimi, osoite, puhelinnumero ja sukupuoli. Koska on mahdollista, että käyttäjä esimerkiksi muuttaa, on tärkeää, että hänellä on mahdollisuus päivittää omia tietojaan. Käyttäjällä on kontrolli omista tiedoistaan, ja yleensä tietoja päivitetään hyvin harvoin, joten profiilisivulla voidaan käyttää strategiana välimuisti turvautuu verkkoon.

Kun käyttäjä ensimmäisen kerran käyttää sovellusta, hänen tietonsa haetaan verkosta ja tallennetaan välimuistiin (esimerkkikoodi 6). Seuraavilla kerroilla käyttäjän tiedot löytyvät välimuistista, joten ne haetaan aina sieltä.

```
self.addEventListener('fetch', (event) => {
    event.respondWith(
        caches.match(event.request).then((response) => {
            return response || fetch(event.request).then((res) => {
                caches.open('mysite-dynamic').then(async (cache) => {
```

```
                    cache.put(event.request, res)
                });
                return res.clone();
            });
        })
    );
    if (event.request.method === 'PUT') {
        event.respondWith(
            fetch(event.request).then((res) => {
                caches.open('mysite-dynamic').then(async (cache) => {
                    cache.put(event.request.url, res)
                });
                return res.clone()
            })
        )
    }
});
```

Esimerkkikoodi 6. Välimuisti, varalla verkko -strategian koodi, jota Service worker käyttää kontrolloidakseen verkkopyyntöjä.

Jos käyttäjä päivittää omia tietojaan, muutokset lähetetään verkkopyynnöllä palvelimelle. Uusia tietoja ei kuitenkaan haeta verkosta ja päivitetä välimuistiin, koska käytettävä strategia estää sen. Tämän takia on kontrolloitava myös verkkopyyntö, joka lähetetään käyttäjän päivittäessä tietojaan. Pyyntö kaapataan Service workerin avulla (esimerkkikoodi 6), ja tiedot päivitetään välimuistiin samalla, kun ne päivitetään palvelimelle.

Muut kuin GET-verkkopyynnöt

Käyttäjän tulee voida lähettää viestejä ja päivittää tietojaan. Viestit lähetetään POST-verkkopyynnöllä ja käyttäjätiedot päivitetään PUT-verkkopyynnöllä. Jotta saadaan luotua mahdollisimman hyvä käyttäjäkokemus, pitää käyttäjän voida suorittaa kyseisiä pyyntöjä myös verkottomassa tilassa. Koska POST- ja PUT-verkkopyyntöjä ei pysty tallentamaan välimuistiin, ne pitää tallentaa selaimen tietokantaan, jos verkkoa ei ole.

Prototyyppisovelluksen POST- ja PUT-verkkopyynnöt tallennetaan selaimen indexedDB-tietokantaan. Jotta tietokantaa voidaan käyttää, se tulee ensin avata. Avauksen yhteydessä annetaan tietokannan nimi ja versionumero. Jos tietokantaa ei ole vielä olemassa, se luodaan automaattisesti avauksen yhteydessä. Jos tietokanta on jo olemassa ja annettu versionumero on korkeampi kuin sen, tietokanta päivitetään uudella versiolla [29]. Prototyyppisovelluksessa tietokannan luonti suoritetaan Service workerissä.

Service workerin puolella verkkopyyntö kaapataan (esimerkkikoodi 7), jotta pystytään kontrolloimaan, mitä tapahtuu, jos verkkoyhteyttä ei ole. Jos verkkoyhteys löytyy, käyttäjälle tarjoillaan palvelimelta saatu vastaus, ja jos verkkoyhteyttä ei ole, verkkopyyntö tallennetaan tietokantaan odottamaan, että verkkoyhteys on palannut.

```
getPostResponse = (req) => {
    return fetch(req.clone()).catch((error) => {
        savePostRequests(req.clone().url, data)
    })
};

self.addEventListener('fetch', (event) => {
    if (event.request.method === 'POST') {
        event.respondWith(
            getPostResponse(event.request.clone())
        )
    }
});
```

Esimerkkikoodi 7. POST-verkkopyynnön kontrollointi Service workerissä.

Samalla kun käyttäjä tekee POST- tai PUT-verkkopyynnön, rekisteröidään taustasynkronointioperaatio. Taustasynkronointioperaatio (esimerkkikoodi 8) pyrkii lähettämään tietokantaan tallennetut POST- ja PUT-verkkopyynnöt, kunnes se onnistuu siinä.

```
self.addEventListener('sync', (event) => {
    if (event.tag === 'sync-tag') {
        event.waitUntil(sendPostToServer())
    }
});
```

Esimerkkikoodi 8. Taustasynkronointioperaatio, joka pyrkii lähettämään verkkopyynnön, kunnes onnistuu siinä.

Service workeriin tulee lisätä vielä tapahtuman kuuntelija, jonka avulla voidaan kontrolloida sivuston puolelta lähetettyjä viestejä (esimerkkikoodi 9). Jos sivuston puolelta tullut viesti sisältää vaadittavat osat, ne tallennetaan muuttujiin. Näitä muuttujia käytetään myöhemmin, jos pyynnöt joudutaan tallentamaan tietokantaan.

```
self.addEventListener('message', (event) => {
  if(event.data.hasOwnProperty('body')&&event.data.hasOwnProperty('headers')){
     body = event.data.body
     headers = event.data.headers
     data = event.data
  }
});
```

Esimerkkikoodi 9. POST- ja PUT-verkkopyyntöjen yhteydessä sivustolta lähetetyn datan kontrollointi tietokantaan tallentamista varten.

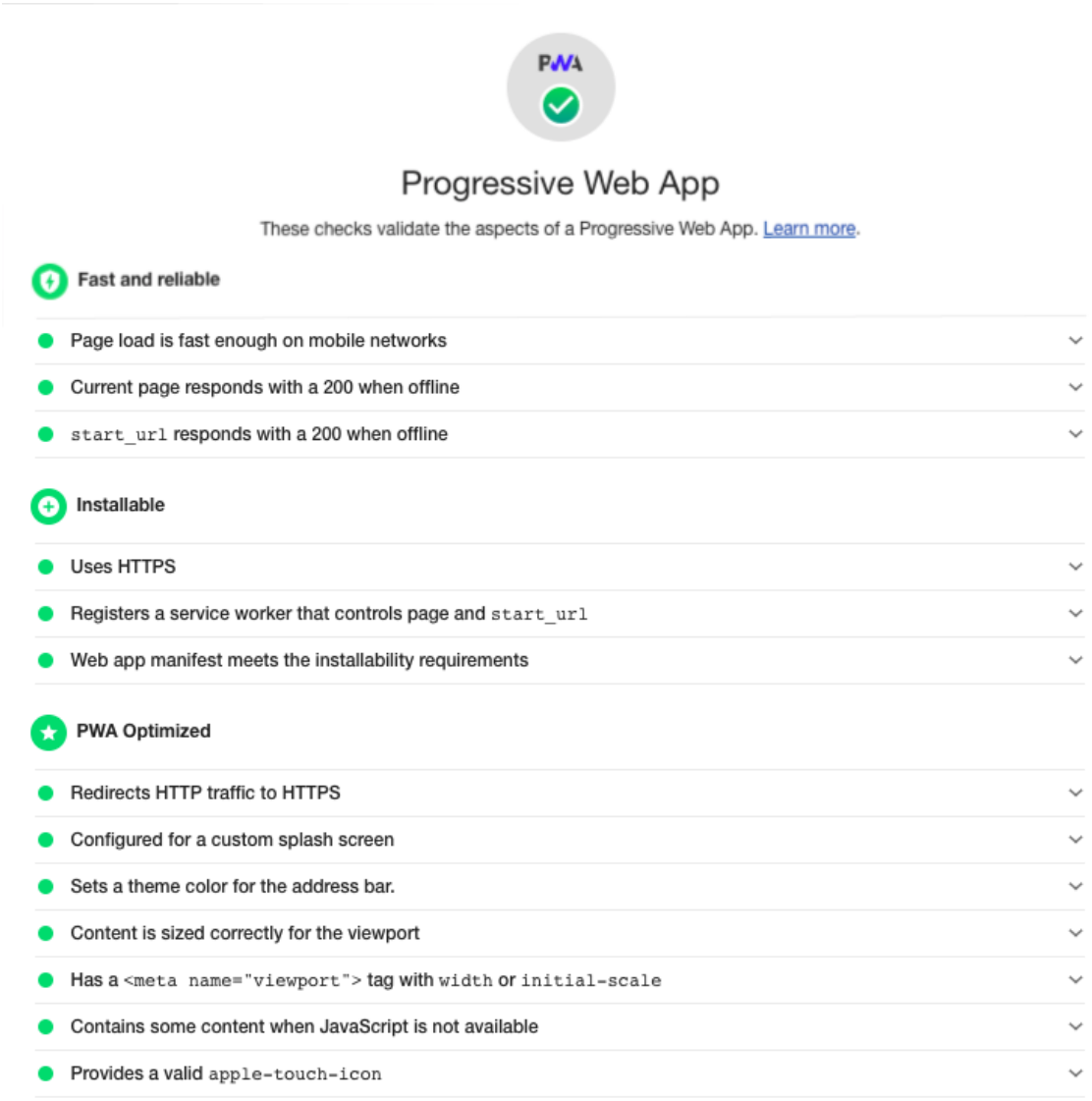
Esimerkkikoodi 10 sisältää sivuston koodin puolella tehtävän POST-verkkopyynnön ja sen yhteydessä rekisteröitävän taustasynkronointioperaation. Lisäksi esimerkkikoodissa näytetään, kuinka pyynnössä käytettävä data lähetetään Service workerille postMessage()-metodilla, jotta se pystyy tallentamaan tiedot tietokantaan, jos verkkopyyntö epäonnistuu.

```
submit = async () => {

    let body = {
        from: 'user',
        message: this.state.messageInput
    }

    let headers = {
        "Content-type": "application/json; charset=UTF-8"
    }

    let msg = {
        'body': body,
        'headers': headers
    }

    navigator.serviceWorker.controller.postMessage(msg)

    navigator.serviceWorker.ready.then((registration) => {
        return registration.sync.register('sync-tag')
    }).then(() => {
        fetch('https://back-opinnaytetyo.herokuapp.com/api/v1/chat', {
            method: 'POST',
            body: JSON.stringify(body),
            headers: headers
        })
    })
};
```

Esimerkkikoodi 10. POST-verkkopyynnön lähettäminen ja taustasynkronointioperaation rekisteröinti.

4.3 Testaus

Prototyyppisovelluksen varmentaminen progressiiviseksi verkkosovellukseksi suoritettiin Lighthouse-työkalulla. Prototyyppisovellus läpäisi kaikki Lighthouse-työkalun asettamat kriteerit (kuva 12), ja näin ollen se sisältää progressiiviselta verkkosovellukselta vaaditut ominaisuudet.



Kuva 12. Prototyyppisovellukselle Lighthouse-työkalulla tehdyn testauksen tulokset.

Prototyppisovelluksen valmistuttua testattiin, että välimuistin käyttöstrategiat oli saatu oikein toteutettua ja sovellus toimisi hitaalla verkkoyhteydellä ja kokonaan ilman verkkoyhteyttä.

Sovellusta testattiin Chromen työpöytäselaimella. Sen avulla saatiin simuloitua hidas verkkoyhteys. Työpöytäselaimella testaaminen suoritettiin palveluntarjoajan näkökulmasta. Sovellusta testasi myös ulkopuolinen henkilö, joka käytti testaamiseen Android-mobiililaitetta. Ulkopuolinen henkilö testasi sovellusta alueilla, joissa on huono ja pätkivä verkkoyhteys. Lisäksi henkilö suoritti testaamista työpaikkansa liikuntatiloissa, jotka sijaitsevat maan alla, jossa ei ole verkkoyhteyttä.

Testausten tuloksena havaittiin, että sovellus toimii ja säilyttää hyvän käyttäjäkokemuksen, vaikka verkkoyhteys olisi huono. Ensimmäisen latauskerran jälkeen sovellusta pystyttiin käyttämään kokonaan ilman verkkoyhteyttä. Sovelluksessa voitiin myös tehdä viestin lähetys tilassa, jossa ei ollut verkkoyhteyttä, ja yhteyden palauduttua viesti lähti sovelluksesta palvelimelle automaattisesti.

5 Yhteenveto

Opinnäytetyössä tutustuttiin progressiivisten verkkosovellusten eri teknologioihin ja selvitettiin, millaisia vaatimuksia sovelluksen tulee täyttää, jotta sitä voidaan kutsua progressiiviseksi verkkosovellukseksi. Lisäksi työssä tutustuttiin erilaisiin välimuistin käyttöstrategioihin ja siihen, kuinka niiden avulla sovellus saadaan toimimaan täysin verkottomassa tilassa.

Työssä oli myös tarkoitus näyttää, kuinka progressiivisten verkkosovellusten tehokkuutta ja helppokäyttöisyyttä voidaan optimoida Lighthouse-työkalun avulla. Työ alkoi kuitenkin laajentua liian suureksi, ja hyvin rajallisen aikataulun vuoksi sen toteutus jouduttiin jättämään pois.

Osana työtä luotiin prototyyppisovellus, jonka tarkoituksena oli havainnollistaa välimuistin käyttöä. Suurimmaksi haasteeksi työssä muodostui tiedon lähettäminen sovelluksesta. Ongelmallisinta siinä oli eri selainten valmius käyttää taustasynkronointia. Taustasynkronointi on tuettu ainoastaan Chrome-selaimella, ja siksi prototyyppisovelluksen testaaminen iOS-laitteilla jätettiin pois.

Kun prototyyppisovellus saatiin valmiiksi, sitä testattiin simuloidun hitaan verkkoyhteyden kanssa palveluntarjoajan näkökulmasta. Lisäksi sovellus annettiin testattavaksi ulkopuoliselle henkilölle, joka joutuu työskentelemään alueilla, jossa on hidas verkkoyhteys tai ei verkkoyhteyttä ollenkaan. Testihenkilön tehtävänä oli käyttää sovellusta kuluttajan näkökulmasta. Testien tuloksena prototyyppisovellus saatiin toimimaan tehokkaasti niin kuluttajan kuin palveluntarjoajan näkökulmasta. Eri välimuistinkäyttöstrategioita yhdistelemällä sen käyttö on mahdollista myös täysin ilman verkkoyhteyttä.

Lähteet

1 Progressive Web App, mistä oikein on kyse? 2017. Verkkoaineisto. Symbio Finland Oy. <https://www.symbio.com/fi/progressive-web-app-mista-oikein-kyse>. 24.5.2017. Luettu 16.12.2019.

2 Haldar, Mahesh. 2018. What is a PWA and why should you care? Verkkoaineisto. <https://blog.bitsrc.io/what-is-a-pwa-and-why-should-you-care-388afb6c0bad>. 1.8.2018. Luettu 16.12.2019.

3 Cantente, Sandro. 2019. What Are Progressive Web Apps (PWAs)? Verkkoaineisto. <https://www.outsystems.com/blog/posts/progressive-web-apps-pwa>. 17.9.2019. Luettu 16.12.2019.

4 Osmani, Addy. 2019. The App Shell Model. Verkkoaineisto. <https://developers.google.com/web/fundamentals/architecture/app-shell>. 14.5.2019. Luettu 3.1.2020.

5 Introduction to Service Worker. 2019. Verkkoaineisto. Google LLC. <https://developers.google.com/web/ilt/pwa/introduction-to-service-worker>. 10.6.2019. Luettu 3.1.2020.

6 Gaunt, Matt. 2019. Service Workers: an Introduction. Verkkoaineisto. <https://developers.google.com/web/fundamentals/primers/service-workers>. 9.8.2019. Luettu 3.1.2020.

7 Archibald, Jake. 2019. The Service Worker Lifecycle. Verkkoaineisto. <https://developers.google.com/web/fundamentals/primers/service-workers/lifecycle>. 8.4.2019. Luettu 3.1.2020.

8 Enyinnaya, Chimezie. 2019. Demystifying The Service Worker Lifecycle. Verkkoaineisto. <https://www.digitalocean.com/community/tutorials/demystifying-the-service-worker-lifecycle>. 12.12.2019. Luettu 3.1.2020.

9 Deveria, Alexis. 2019. Service Workers. Verkkoaineisto. <https://caniuse.com/#search=service%20worker>. Luettu 25.1.2020.

10 Web App Manifest. 2019. Verkkoaineisto. Mozilla. <https://developer.mozilla.org/en-US/docs/Web/Manifest>. 19.1.2020. Luettu 25.1.2020.

11 LePage, Pete. 2019. Add to Home Screen. Verkkoaineisto. <https://developers.google.com/web/fundamentals/app-install-banners/#criteria>. 30.7.2019. Luettu 3.1.2020.

12 Deveria, Alexis. 2019. Web App Manifest. Verkkoaineisto. <https://caniuse.com/#search=web%20app%20manifest>. Luettu 25.1.2020.

13 Archibald, Jake. 2019. Introducing Background Sync. Verkkoaineisto. <https://developers.google.com/web/updates/2015/12/background-sync>. 14.1.2019. Luettu 3.1.2020.

14 Deveria, Alexis. 2019. Background Sync API. Verkkoaineisto. <https://caniuse.com/#search=background%20sync>. Luettu 25.1.2020.

15 Introduction to Push Notifications. 2019. Verkkoaineisto. Google LLC. <https://developers.google.com/web/ilt/pwa/introduction-to-push-notifications>. 1.5.2019. Luettu 3.1.2020.

16 Why Are Push Notifications So Important? 2018. Verkkoaineisto. RubyGaraga. <https://rubygarage.org/blog/benefits-of-push-notifications>. 11.6.2018. Luettu 25.1.2020.

17 Lasn, Indrek. 2019. Everything You Need to Know About PWAs — Push Notifications. Verkkoaineisto. <https://medium.com/better-programming/everything-you-need-to-know-about-pwas-push-notifications-e870bb54e14f>. 28.6.2019. Luettu 25.1.2020.

18 Deveria, Alexis. 2019. Notification API. Verkkoaineisto. <https://caniuse.com/#search=notification>. Luettu 25.1.2020.

19 Deveria, Alexis. 2019. Push API. Verkkoaineisto. <https://caniuse.com/#search=push%20notification>. Luettu 25.1.2020.

20 Novicki, David. 2019. PWA Asset Caching Strategies. Verkkoaineisto. <https://codeburst.io/pwa-asset-caching-strategies-8a20c31b2181>. 29.1.2019. Luettu 4.1.2020.

21 Archibald, Jake. 2019. The Offline Cookbook. Verkkoaineisto. <https://developers.google.com/web/fundamentals/instant-and-offline/offline-cookbook>. 12.2.2019. Luettu 4.1.2020.

22 Cache only. Verkkoaineisto. Mozilla. <https://serviceworke.rs/strategy-cache-only.html>. Luettu 4.1.2020.

23 Progressive Web Apps: Caching Strategies. 2019. Verkkoaineisto. DEV Community. <https://dev.to/mr_steelze/progressive-web-apps-caching-strategies-mf2>. 20.1.2020. Luettu 25.1.2020.

24 Aryan, Avi. Stale-while-revalidate Data Fetching with React Hooks: A Guide. Verkkoaineisto. <https://www.toptal.com/react-hooks/stale-while-revalidate>. Luettu 4.1.2020.

25 Posnick, Jeff. 2019. Keeping things fresh with stale-while-revalidate. Verkkoaineisto. <https://web.dev/stale-while-revalidate>. 18.7.2019. Luettu 4.1.2020.

26 React. Verkkoaineisto. Facebook Inc. <https://reactjs.org>. Luettu 16.12.2019.

27 Pati, Chinmaya. 2019. Custom Service Worker in CRA(Create React App). Verkkoaineisto. <https://medium.com/@chinmaya.cp/custom-service-worker-in-cra-create-react-app-3b401d24b875>. 22.10.2019. Luettu 16.12.2019.

28 Material-UI. Verkkoaineisto. Material-UI. <https://material-ui.com>. Luettu 16.12.2019.

29 Hiwarale, Uday. 2018. How to use IndexedDB to build Progressive Web Apps. Verkkoaineisto. <https://itnext.io/indexeddb-your-second-step-towards-progressive-web-apps-pwa-dcbcd6cc2076>. 4.4.2018. Luettu 25.1.2020.